Software Recapitalization Economics

David Lechner
WeC2 Technologies

This article analyzes the economics of cyclic replacement or recapitalization of software. It analyzes some of the historic issues and costs of software maintenance. Then, it provides analysis on how much software modernization occurs while still maintaining old code. I recommend a combination of minimal maintenance and a cyclic re-engineering to maximize the amount of code refresh and minimize the code's average age. In theory, regular refresh will allow rapid introduction of new or improved functions. It finishes by recommending strategies to target improved productivity as a component of recapitalization.

Large commercial and government systems are increasingly software intensive. The costs of supporting software over a 10 to 20-year lifetime are increasingly significant in today's software dependent world. Software support costs continue long after development ends, typically costing between 67 percent and 80 percent of the overall life-cycle cost. As shown in Figures 1A and B [1, 2], this is two-to-four times the development cost. This article will investigate options on how to spend that life-cycle maintenance budget to both maintain and enhance the code at the same time.

The U.S. Air Force (USAF) [2] studied 487 commercial software development organizations to see how software support costs are distributed among different tasks. According to the report, most software support dollars are spent on defining, designing, and testing changes. Support activities they identified include the following:

- Interacting with users to determine what changes or corrections are needed.
- Reading existing code to understand how it works.
- Changing existing code to make it perform differently.
- Testing the code to make sure it performs both old and new functions correctly.
- Delivering the new version with sufficient new documentation to support the user/product.

The USAF also shows that in post-deployment software support, 75 percent of the effort involves *enhancement* and refinement, and the remaining 25 percent is associated with *maintenance* of existing modules, as shown in Figure 2 (see page 22) [2]. Historically, the maintenance of software involves correcting bugs and supporting the required deployment changes. Research by Roger Pressman in *Software Engineering: A Practitioner's Approach* also confirmed this 21 percent factor and found that:

> … only about 20 percent of all support work is spent fixing mistakes, while the remaining 80 percent is spent adapting existing systems to changes in their external environment, making enhancements (possibly categorized as refinement) requested by users, and reengineering an application for future use. [4]

We will use the term *enhancement* for all non-maintenance program improvements, including refinement. We will use the term post-deployment support to include both maintenance and enhancement.

According to the USAF and other research, newly developed software has a high failure rate until the bugs are worked out, and after this point, failure rates usually drop to a low level [1, 2]. Theoretically, software should stay at that *reliable* level forever because it is not getting physical wear. Since software continues to get changed, however, it will continue to have bugs and reliability problems. *Thus, software wears out because it is maintained* [5].

There are more modern reasons for software maintenance, namely the following:

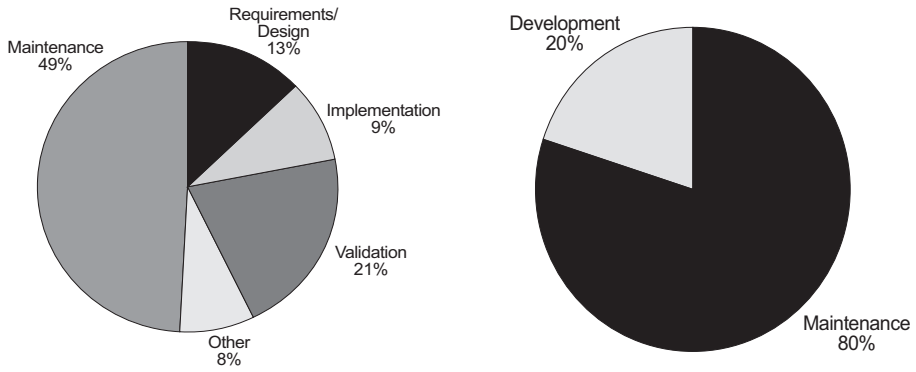
- Changes required for security considerations and protection from hackers.
- Changes in the operating system (versions), often driven by security or reliability.
- Changes due to other changing commercial off-the-shelf (COTS) software or hardware products.

Validating the Maintenance Expense

The 2:1 relationship of software support cost to development costs (i.e. 66 percent to 33 percent ratio) is critical to developing a strategy for life-cycle maintenance, but how real are these *typical* relationships? One significant study published data on coding errors and support [6]. The relevant data in Table 1 (see page 22) was taken from the Information Technology department of a large commercial bank. This data represented 35 projects and 20 million source lines of code (SLOC). We tested those rules of thumb by using them to calculate the theoretical support costs and compared them to the actual reported annual support cost.

The *age* of code was not available, so we assumed a mean 30 year lifetime and a *typical* productivity of one line of code per hour at a cost of $50 per man hour to obtain a derived development cost. That derived development cost was then multi-

Figures 1 A and B: *Software Life-Cycle Support Cost for Two Types of Large Programs*



Note: The USAF report used the term *maintenance* for generic post-deployment life-cycle support [2].

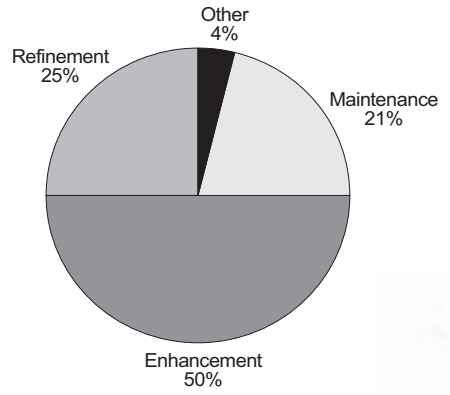


Figure 2: *Approximately 75 Percent of Software Post-Deployment Support Is Associated With Refinement and Enhancement to the Program* [3, 2]

plied by two and divided by 30 years to get a theoretical annual support cost, which we compared to the actual expenditures by the bank. This ratio was within 3 percent for the mean project value and 7 percent for the average project. This calculation does not correct for the present value versus annual value of money or the actual age of each program. The *rules of thumb* on software life-cycle maintenance however still seem reasonable and applicable today.

Given that the life-cycle software support costs is twice the development cost, we can divide that by a nominal 20 years lifetime to estimate annual support costs. We use 20 years here instead of 30 years since technology is moving quickly and we want a conservative estimate. This ratio implies that support costs 10 cents per year for each development dollar spent. This annual expense does not cover the capital recovery of the initial development expenses. We call this the *10 percent rule for software maintenance budgeting*. This rule checks the adequacy of a project's long-term software support budget, or perhaps determines the required sales/licensing revenue to justify a commercial project. To have a really successful long term project, with functionality growth and satisfied users (and perhaps a sustaining business operations point), the ratio of a long term support budget (or revenue) vs. a development budget should be at or above 10 percent. To recover development costs *and* do maintenance, the product income stream must be even higher (with a three year payback on development costs, the annual revenue must exceed 40 percent of the development budget).

This ratio illustrates the problem with starting up new software projects on a regular basis (such as Department of Defense Small Business and Innovative Research projects) unless they replace old code. The project is a distraction that siphons budgets from other maintenance needs unless the customer is prepared to spend 10 percent of the development cost on an annual basis to support the code. If the vendor does not commercialize to recoup the difference, the project is doomed.

This 10 percent rule can be dissected further. Data shows that 25 percent of costs are for maintenance and *other*, and 75 percent of the long term cost is for *refinement* and *enhancement* [3]. Thus out of our 10 percent of development expense, there is an annual 2.5 percent minimum budget required for reliability and bug fixing and 7.5 percent recommended for functionality change.

The High Costs of Software Support

Unfortunately, software support cost for old code, measured as a ratio per line of source code, can be up to 40 times more expensive than engineering new code [1]. This factor has a huge impact on long-term software support. It is commonly attributed to several reasons, namely the following:

1. The support engineers are often not the original developers and must relearn the program's design and requirements.
2. The code becomes more complex as it evolves and is patched, introducing more bugs and requiring more testing [7].
3. The programming technology is old and static, requiring greater labor.
4. Documentation problems require additional effort and repair.
5. There are definitive system limitations on how fixes can be made.
6. Extensive testing to ensure that the system performance is met [8, 9].
7. The use of older technology requiring older (more senior and more expensive) workers, often considered *gurus*, who are the only ones with the familiarity that can make a change [10].

Typical legacy style support involves analyzing user problems and requests, operating system updates, COTS product version changes, urgent reliability problems and security issues, and hardware obsolescence issues in order to define and determine what support is done. This effort can be challenging since some problems are incredibly difficult to find or recreate. Teams supporting legacy code typically work in a cyclic mode based on budget availability.

Software Re-Engineering

Software re-engineering is the complete overhaul of a software application, tearing it down to its component requirements and rebuilding it with modern methods, codes, and practices. The USAF policies advocate re-engineering of software *when the program manager concludes that it is better to pay now, rather than waiting to pay much more later* [11]. *Paying now* is what William E. Perry calls *avoiding the rathole syndrome*. Perry defines a rathole as *the dark place where software maintainers throw their money with no possibility of return on investment*, and he compares software with old cars. In the short term, it is cheaper to fix your old car than it is to buy a new one, but in the long term it costs more than buying a new car. Perry also explained that once one software rathole is plugged up (bug is fixed) another usually appears.

The 40:1 ratio of costs for maintaining old vs. new code should really be a factor that changes over time, starting out closer to a regular 1:1 factor if the maintenance team was chosen from the remnants of the original developers and used the same development environment. It would then worsen as the staff changes, tools age,

Table 1: *Recent Commercial Data Helps Validate That Post-Deployment Life -Cycle Costs Are Twice Development Costs*

Commercial Application Systems Profile (35 Systems, Total 20M SLOC, 1970s to 1990s) [6]				
Variable	Mean	Average	Minimum	Maximum
Errors per Month (After Deployment)	8.3	9.7	1	101
Support Costs/Year	$693,000	$661,000	$83,000	$3,532,000
Application Size – in # of Modules	217	266	18	1,500
Application Size – Lines of Code	215,000	185,000	54,000	702,000
Authors Analysis (Added):				
Rule of Thumb Development Cost	$10,750,000	$9,250,000	$2,700,000	$35,100,000
30 Years of Support Would Then Cost	$20,790,000	$19,830,000	$2,490,000	$105,960,000
Percentage Support Cost of Total Life Cycle Cost	66%	68%	48%	75%
Dev. Cost * (2/ Life)	$716,667	$616,667	$180,000	$2,340,000
Ratio of Theoretical to Actual Support Cost	103%	93%	217%	66%

complexity increases, and reliability degrades due to change [8, 9]. Some researchers identify code that is over 15 years old as *alien* code [8, 9], since by this age there are no longer any members of the original development staff left. Today, most applications that are old are unstructured, have architecture problems, poor documentation, and questionable change records. A good support strategy should avoid letting code get too old.

The *average equipment age* is a concept used in economics, replacing a capital item on a regular cycle is called *recapitalization*. At any given time the average age is the sum of the different ages multiplied by a weighting factor that is the inverse of the cyclic time period. In the example below, we shortened the math using the formulae for a series summation developed by Carl F. Gauss in the 1800s. The average age of an inventory just before one unit is replaced is simply half of the sum of one plus the cycle time period.

7 Year Cycle: (1/7)*1 + (1/7)*2 +(1/7)*3 + (1/7)*4 +(1/7)*5 + (1/7)*6 + (1/7)*7
= (1/7) * {1+2+3+4+5+6+7 }
= (1/N) * {Summation 1:N}

Generalized Case; Average Age of Items Replaced Every N Years:
= (1/N)*{N * (N+1)/2} by using Gauss' summation formula
= (N+1) / 2

Can we apply a cyclic recapitalization strategy to get code improvement via re-engineering and still continue acceptable support? To answer this question we analyzed a large hypothetical project of 20 million source lines of code. The key consideration is the amount of new software source code (new SLOC) created each year. One key input is the mix or allocation between maintained old code and newly *re-engineered* code. The other is the productivity, which ranged from eight SLOC/day for new code to 0.2 SLOC/day for old code (40 times worse). Software productivity is actually a very complex product of many factors besides the age of the code [12]. There are many problems with simple productivity factors, but we ignore them to seek a simple approximation and show hypothetical first-order effects based on the age of the code.

We calculate the cost of software code by multiplying the daily labor rate ($50/New SLOC at $100,000 per man year) by the coding productivity (equivalent new SLOC per day). Our simple labor rate allows the readers to easily scale their own labor costs.

ASSUMPTIONS / INPUT DATA		
Labor Cost/MY	$100,000	
Productivity: New SLOC/MY	2,000	(8 SLOC/Day)
Development Cost $/SLOC	50	$/SLOC
SLOC Developed	20,000,000	New SLOC
Total Dev $ (SLOC * Cost/SLOC)	$1,000,000,000	($1B)
Support Factor (40:1 more expensive)	40	
Maint. Software Cost $/SLOC	$2,000	#/SLOC
Life	20	Years
Annual $/Year (10% of Development $)	$100,000,000	
OPTION 1: 100% Maintenance Approach		
Maintained SLOC/Year (Based on Annual $/Software Support Cost)	$100,000,000/ ($2,000 /SLOC) = 50,000	SLOC
% SLOC Maintenance/Year =	50K/20M = 0.25%	
(Clearly you did not get much for the $100M Budget)		
OPTION 2: 100% Effort Is Re-Engineering Each Year		
Maintained SLOC/Year (#SLOC) if 100% Dev. Type Work:	$100,000,000/ ($50/SLOC) = 2,000,000	SLOC
% of Program Enhanced =	10%	

Table 2: *Sample Analysis With Post-Deployment Support Options for a Large Program*

We also do not attempt to quantify considerations in the cost of code since it is beyond the scope of this analysis and well covered by other research [13]. Option 1 in Table 2 shows the results of a typical support project, with the heroic efforts invested in supporting old code. As time goes by and the code ages, this investment will only produce 0.25 percent of *changed* code (50,000 lines) per year due to the poor productivity factor. If all of the investment goes into re-engineering code (i.e. replacing an old module with a brand new one), at the *good* productivity rate of eight SLOC/day, the result would be 2,000,000 source lines being modified as shown in Option 2 of Table 2. Clearly changing 10 percent of the program per year provides a good return on the investment and stays far back from the 15-year rathole age identified by [9].

The equation of interest:

$$\text{New SLOC/Yr} = [B_A * \%RE * (E_{RE} / C_{labor})] + [B_A * \%M * (E_M / C_{labor})]$$

where

B_A = Annual SW Maintenance Budget, in $/year
%RE = Percentage Budget Spent on Re-engineering,
E_{RE} = Efficiency for Re-engineering, (units as SLOC/$)
C_{labor} = Cost of Labor, Assumed Constant
%M = Percentage of Budget Spent on Legacy Maintenance, and
E_M = Efficiency for Maintenance Efforts (Units as SLOC/$)

Note that the amount of code is dependent on the cost per SLOC which is an extremely complex factor. It can even vary inversely to expectations, with newer projects often costing less overall but having a much greater cost per source line of code due to having used compact 4GL languages [12]. Our simple metrics are therefore starting points which allow the reader to make comparisons with their own project or metrics.

Figure 3 (see page 24) shows the results of a sensitivity analysis where we vary the percent of investment into re-engineering and the productivity ratio. The bar heights represent how much total code is changed each year. A proven numeric relationship for the productivity factor as a function of code age would have improved this model but was unavailable.

The desired position to be on this graph is the *back corner* where the team productivity is high (i.e. low ratio) and the percent of code being re-engineered on a planned cycle is high (lots of new functionality possible at lower costs). The result would be newly re-engineered modules that have a lot of the new capa-

bilities that the users want, albeit released at a slow, cyclic pace. Users may be asking to delete functions, add new ones (more code) or just change how the old one works (in theory not adding code).

The product manager or customer liaison team would have to balance the added functionality and deleted features within the annual budget. It would seem prudent however to assume some marginal loss in productivity in determining the amount of code changed each year and allow users to add new capabilities. Hopefully customers and *users* of a software program would be satisfied and *live with* some problems as long as others are getting fixed. Some of the reasons stated for *maintenance* (e.g. security and reliability) are pretty imperative, so it is unlikely that the maintenance can be completely eliminated. Earlier we discussed a target of 25 percent of the annual budget as being necessary for this normal maintenance activity.

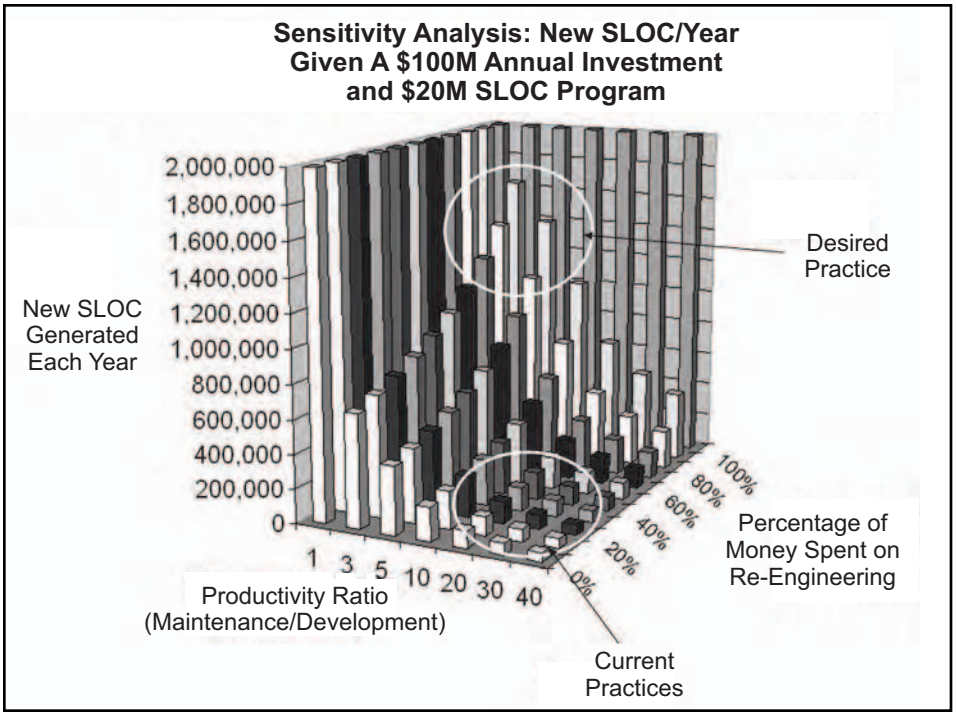
The balance of a *fully funded* annual support budget (that annual 10 cents per initial development dollar) would be 75 percent of the support budget available for *new* functionality via re-engineering. Here we are considering percentages of support dollars however, not percentages of SLOC that comes from including productivity. A 10-year replacement cycle would have an average age of just over five years. It avoids getting into the *rathole* region at 15 years and appears achievable with an allocated mix of 25 percent spent on true maintenance and 75 percent for enhancement via re-engineering.

Some recent research has indicated that the bulk of the maintenance effort gets concentrated on only 20 percent of the program code [14]. The maintenance efforts may indeed focus on a small subset of code, but this does not mean that 80 percent of the code does not ever need re-engineering. Re-engineering provides the users with new functionality and modernizes the code. Thus, managers need to examine systematic replacement of the whole code base in order to keep the average age low and avoid ratholes.

The challenge of improving efficiency in software maintenance is considerable. Our analysis assumed a linear worsening of productivity with age, but in reality productivity rates will need to be aggressively managed to be successful. There are several strategies that could be used to do this, namely the following:

- **Use modern tools.** We have shown numerically that cyclic replacement and minimized average code age can have positive effects. Most large, old programs have both old and new modules with ready candidates for replacement, but even newly completed projects must start aggressively replacing modules since a permanent bias will creep into the average age for each year of delay.
- **Modular software design.** This is the basis of object-oriented design principles, and benefits the code maintainer by allowing easier replacement of modules. The method called software *re-factoring* advocates replacing software on a modular basis as developed by Ward Cunningham [15] and Martin Fowler [16]. A modular strategy may help maintenance by providing defined interfaces and functionality.
- **Add more off-the-shelf code.** This leverage of existing code or products can save support cost, but this can also cause problems. Some vendors change their product or stop supporting a version, thus increasing maintenance costs. Databases, publish/subscribe services, and web servers and browsers are all modern examples of products that can modernize code.
- **Staff rotation.** Regular rotation of staff between development and maintenance teams could minimize the differences in skill sets, productivity, experiences, and employee enthusiasm. Engineers that have maintenance experience know first-hand why it is important to develop code for maintenance.
- **Modern languages.** New software languages that use fewer lines of code to do the same function can help lower support costs. Alan Albrecht at IBM developed methods to use function points as a measure of software functions to estimate program size [17]. Current data shows that HTML and Web-service methods take 25 percent less SLOC than C+ or Java for similar function points (functionality) [18]. Another study [19] showed that a representative program of 300 Web objects (function points, links, multimedia files, scripts and Web building blocks) took 38 percent fewer person-months to develop when HTML was used instead of Java. This implies that added emphasis should be placed on replacing older code with newer software structures and languages in order to capture the improved productivity rate.

Figure 3: *The Amount of Code Changed as a Function of the Mix of Maintenance Investment and Productivity Factors*



Concluding Remarks

Software support is expensive, is absolutely necessary, and will be necessary for many years. In military programs, the support costs are borne by the taxpayers, and in commercial projects, support expense is subtracted from sales or licensing revenue. Strategies for cyclic recapitalization of the code to keep the average age young and improve programming productivity should minimize long-term maintenance costs and allow the support team to regularly add improve-

ments and functionality.◆

References

1. Boehm, Barry W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.
2. United States Air Force. Software Technology Support Center. Guidelines for Successful Acquisition of Software Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems (GSAM). Version 3.0 May 2000 <www.stsc.hill.af.mil/resources/tech_docs/>.
3. Piersall, COL James. "The Importance of Software Support to Army Readiness." Army Research, Development, and Acquisition Bulletin. Jan.-Feb. 1994.
4. Pressman, Roger S. Software Engineering: A Practitioner's Approach 3rd ed. New York, NY: McGraw-Hill, 1992.
5. Glass, Robert L. Building Quality Software. Englewood Cliffs, NJ: Prentice Hall, 1992.
6. Banker, Rajiv, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software Errors and Software Maintenance Management. The Netherlands: Kluwer Academic Publishers, 2002.
7. Lehman, M.M. "Programs, Life Cycles, and Laws of Software Evolution." Proc. of the IEEE 68.9 (Sept. 1980): 1060-1076.
8. Eisenbach, Susan. "Software Maintenance, No Jokes, Lectures 2 and 3." <www.doc.ic.ac.uk/~sue/475/lec%202%202%20&%203%20-%20maintenance.pdf>.
9. Lano, Kevin and Howard Haughton. Reverse Engineering and Software Maintenance. New York: McGraw Hill, 1994.
10. Wade, Stu and Andy Laws. Legacy System Management via the Triage Model, Software Triage. Liverpool, UK: John Moores University, 1998 <www.cms.livjm.ac.uk/research/docs/CMS1898.DOC>.
11. Perry, William E. "Don't Pour Money Down Rat Holes that Infest Your Budget." Government Computing News Dec. 1993.
12. Jones, Capers. Programming Productivity. New York: McGraw Hill, 1986.
13. Hinh, Jarus M. "The Impact of Faster, Better, Cheaper." Spacecraft Ground Systems Architecture Workshop, 1999 <http://sunset.usc.edu/events/GSAW/gsaw99/dpdf-presentations/breakout-2/hihn-1.pdf>.
14. Boehm, Barry W. "The Economics of Software Reliability." International Symposium on Software Reliability Engineering, 19 Nov. 2003 <www.cs.colostate.edu/~malaiya/issre/barryboehm.pdf> .
15. Beck, Kent. eXtreme Programming eXplained: Embrace Change. Reading, MA: Addison Wesley Longman, 1999.
16. Fowler, Martin, et. al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
17. Albrecht, Alan J. "Measuring Application Development Productivity." Proc. of the Joint SHARE/GUIDE/IBM Application Development Symposium, Oct. 1979.
18. Quantitative Software Management, Inc. "Function Point Programming Languages Table." Version 2.0 2002 <www.qsm.com/FPGearing. html>.
19. Reifer, Donald J. "Estimating Web Development Costs: There Are Differences." CROSSTALK June 2002 <www.stsc.hill.af.mil/crosstalk/2002/06/reifer.html>.

About the Author

David Lechner currently works as a consultant for GeoLogics Corp., and recently founded WeC2 Technologies, a small business focusing on Web-enabled C2 software. He has 20 years of experience in Department of Defense systems development and acquisition. Lechner's civil service career includes: Naval Air Systems Command (Reconnaissance, Electronic Warfare, Special Operations, Navy); Naval Electronic Systems Command (PMW143); General Systems Administration (Federal Systems Integration and Management Center); and Naval Sea Systems Command (PMS425). His private industry experience includes DRS Electronics Systems and GeoLogics Corp. Lechner has a bachelor of science in Electrical Engineering from Carnegie Mellon University, a M.E.Ad. from George Washington University, and a master of science in Computational Physics from George Mason University.

GeoLogics Corp.
5285 Shawnee RD
STE 300
Alexandria, VA 22312
Phone: (240) 418-1544
Fax: (703) 750-4010

COMING EVENTS

November 30-December 8
QFD 2006
18th Symposium on QFD
Austin, TX
www.qfdi.org/symposium.htm

December 3-8
LISA '06
20th Large Installation System Administration Conference
Washington, D.C.
www.usenix.org/events/lisa06

December 4-7
ICSOC 2006
4th International Conference on Service Oriented Computing
Chicago, IL
www.icsoc.org/

December 4-7
IITSEC 2006 Interservice/Industry Training Simulation and Education Conference
Orlando, FL
www.iitsec.org/index.cfm

December 5-7
XML 2006
Boston, MA
http://2006.xmlconference.org

December 11-15
ACSAC 2006 Annual Computer Security Applications Conference
Miami, FL
www.acsac.org

December 14-15
11th Annual ITC East 2006 Conference
Harrisburg, PA
www.govresources.com/itceast2006.html

June 18-21, 2007
2007 Systems and Software Technology Conference
Tampa, FL
www.sstc-online.org